



Open Architecture Technical Principles and Guidelines 1.5.8

*Author: Eric M. Nelson
ericnels@us.ibm.com
Owner: IBM Federal CTO Office*

Document History

Document Location

This is a snapshot of an on-line document. Paper copies are valid only on the day they are printed. Refer to the author if you are in any doubt about the currency of this document.

Revision History

Date of this revision: 02-01-07	Date of next revision <i>(date)</i>
---------------------------------	-------------------------------------

Revision Number	Revision Date	Summary of Changes	Changes marked
1.0	02-01-07	First Draft	N
1.2	02-05-07	Modifications based on feedback from Fred Mervine and Peter Bahrs. Added section 3.1 to identify key criteria under which using OA is necessary, and where it is not relevant.	Y
1.3	03-09-07	Added OA Reference Model and business driver analysis.	N
1.4	03-09-07	Added analysis of OA Principle application to TCO	N
1.5	03-30-07	Modifications based on feedback from Jeb Brown and Richard Ernst	N
1.5.2	04-10-07	Minor edits, added attribution, some minor text additions	N
1.5.3		Not yet posted version	N
1.5.4	9-19-07	ITAR Approval Notification Added	N
1.5.5	11-15-07	Minor edits	N
1.5.6	3-24-08	Extensive edits, esp. in section 2.2 and Section 5	N
1.5.7	6-30-08	Additions to enablers and inhibitors	N
1.5.8	9-30-08	Minor edits	N

Approvals

This document requires following approvals. Signed approval forms are filed in the Quality section of the PCB.

Name	Title
<i>(name)</i>	<i>(title)</i>

Distribution

This document has been distributed to

Name	Title
<i>(name)</i>	<i>(title)</i>

Contents

Document History	2
Document Location	2
Revision History	2
Approvals	2
Distribution	2
Contents	3
1. Introduction	7
1.1 Principles and Guidelines	7
1.2 Open Architecture	8
1.3 OA Business and Technology Drivers	8
1.4 OA and SOA (Service Oriented Architecture).....	9
1.4.1 OA Principles essential to SOA	9
1.4.1.1 Modularity	9
1.4.1.2 Open Standards	10
1.4.1.3 Interoperability	10
1.5 Document Scope.....	10
1.6 Related (future) Open Architecture White Papers	10
1.7 References.....	10
1.8 Acknowledgements.....	11
2. Open Architecture Reference Model.....	12
2.1 Open Architecture Technical Nonfunctional Requirements.....	12
2.1.1 Open Standards.....	13
2.1.1.1 Definition.....	13
2.1.1.2 Dependencies on other OA NFRs.....	14
2.1.2 Modularity	14
2.1.2.1 Definition.....	14
2.1.2.2 Dependencies on other OA NFRs.....	14
2.1.3 Interoperability	14
2.1.3.1 Definition.....	14
2.1.3.2 Dependencies on other OA NFRs.....	15
2.1.4 Extensibility	15
2.1.4.1 Definition.....	15
2.1.4.2 Dependencies on other OA NFRs.....	15

2.1.5	Reusability	15
2.1.5.1	Definition.....	15
2.1.5.2	Dependencies on other OA NFRs.....	16
2.1.6	Composability	16
2.1.6.1	Definition.....	16
2.1.6.2	Dependencies on other OA NFRs.....	16
2.1.7	Maintainability	17
2.1.7.1	Definition.....	17
2.1.7.2	Dependencies on other OA NFRs.....	17
2.2	Open Architecture Drivers.....	17
2.2.1	Total Cost of Ownership	17
2.2.1.1	Acquisition	18
2.2.1.2	Vendor Payments	18
2.2.1.3	Training.....	19
2.2.1.4	Development/Integration	19
2.2.1.5	Testing.....	20
2.2.1.6	Maintenance	20
2.2.1.7	End of Life.....	20
3.	OA Technical Principles.....	22
3.1	Use Open Standards	22
3.1.1	Use Open Standards Principle Statement.....	22
3.1.2	Motivation for Open Standards Use	23
3.1.3	Implications of Using Open Standards	23
3.2	Use Modular Design	23
3.2.1	Use Modular Design Principle Statement.....	24
3.2.2	Motivation for the use of Modular Design	24
3.2.3	Implications of using Modular Design.....	24
3.3	Design for Interoperability	25
3.3.1	Design for Interoperability Principle Statement	26
3.3.2	Motivation for designing for Interoperability.....	26
3.3.3	Implications of designing for Interoperability	26
3.4	Design for Extensibility.....	26
3.4.1	Design for Extensibility Principle Statement.....	27
3.4.2	Motivation for designing for Extensibility	27
3.4.3	Implications of designing for Extensibility	27

3.5	Exploit Reusability.....	27
3.5.1	Exploit Reusability Principle Statement.....	28
3.5.2	Motivation for exploiting Reusability.....	28
3.5.3	Implications of exploiting Reusability.....	28
3.6	Design for Composability.....	29
3.6.1	Design for Composability Principle Statement.....	29
3.6.2	Motivation for designing for Composability.....	30
3.6.3	Implications of designing for Composability.....	30
3.7	Design for Maintainability.....	30
3.7.1	Design for Maintainability Principle Statement.....	31
3.7.2	Motivation for designing for Maintainability.....	31
3.7.3	Implications of designing for Maintainability.....	31
4.	Related Nonfunctional Requirements.....	32
4.1	Scalability.....	32
4.2	Replaceability.....	32
4.3	Portability.....	32
4.4	Supportability.....	32
4.5	Affordability.....	32
4.6	Information Assurance (IA).....	33
5.	Open Architecture Guidelines.....	34
5.1	Open Architecture Criteria.....	34
5.1.1	OA is needed if.....	34
5.1.2	OA is not needed if.....	34
5.1.3	“Degrees of Openness”.....	35
5.2	About OA Enablers and Inhibitors.....	35
5.3	Organizational Practices.....	35
5.3.1	Enablers.....	36
5.3.2	Inhibitors.....	38
5.4	Open Standards.....	39
5.4.1	Enablers.....	39
5.4.2	Inhibitors.....	39
5.5	Modular Design.....	39
5.5.1	Enablers.....	39
5.5.2	Inhibitors.....	40
5.6	Maintainability.....	40

5.6.1	Enablers.....	40
5.6.2	Inhibitors	40
5.7	Interoperability.....	41
5.7.1	Enablers.....	41
5.7.2	Inhibitors	42
5.8	Extensibility	42
5.8.1	Enablers.....	42
5.8.2	Inhibitors	43
5.9	Reusability.....	43
5.9.1	Enablers.....	43
5.9.2	Inhibitors	43
5.10	Composability	43
5.10.1	Enablers.....	43
5.10.2	Inhibitors	44

1. Introduction

This document describes the Architectural Principles and Guidelines that underlie systems that use an Open Architecture approach to software and systems engineering.

1.1 Principles and Guidelines

Architectural Principles are broad constraints on all software and system design and development. The reason for specifying Principles is to drive architectural, design, development and system management practices that realize business goals and to avoid errors that will limit the success of the system.

Principles differ from Guidelines in that Principles are mandatory. All systems and components developed for an enterprise must adhere to applicable Principles. Exceptions are possible, but they must be carefully justified, reviewed and approved at the enterprise level. The justification for an exception must demonstrate why the Principle(s) cannot be satisfied, how the alternate approach fulfills the intention of the Principle(s), and the risk mitigation actions that will be used to avoid any problems that served as the original motivation.

The barrier for an exception to a Principle must be very high, because it applies to the system at an enterprise level, and any component under review should be, in part, crafted to optimize processes or capabilities across the organization. This means, in particular, that optimizing performance (of any kind) of a specific system, application or component will almost never be a sufficient reason to violate a principle.¹

Guidelines identify best practices and suggestions for all phases of a system lifecycle. Guidelines sometimes trace back to Principles as directions or suggestions for implementation. Other Guidelines may be independent and may reflect lessons learned by the organization or from the broader Communities of Interest.

Principles and Guidelines are fundamental statements of enterprise priorities that must be owned and maintained as part of the core Enterprise Architecture governance processes of an organization. If there is not active review of systems against the principles and guidelines, then they will rapidly be ignored under the pressures of cost and schedules. In the short term, there is always an expense to “having principles” and if there is not an organizational counterbalance to budget and schedule pressures, short term pressures will win out.

Further, a regular reassessment of the validity of the principles in light of technological changes is necessary as part of the governance processes. Principles and guidelines invariably include assumptions, tacit or otherwise, of what is possible with current technology. Change can invalidate those assumptions. If principles and guidelines are not reviewed in light of a changed technical ecosystem, then they will rapidly become irrelevant and if still enforced will become a major problem in the organization. Guidelines in particular should be reviewed because they often explicitly discuss how to realize a principle with current technology.

¹ Claims of performance or other types of exceptions are frequently based on overgeneralization of lessons learned in the use of prior technology or processes. Generally these should be viewed with suspicion. But, for example, if performance is key to a safety-critical system or to a component that has clear deterministic real-time requirements, then a well-partitioned portion of a system may be approved for an exception to a principle that makes it impossible for it to meet its performance requirement. This still requires careful analysis, a clear justification and an effort to minimize the amount of code/hardware that falls under the exception.

1.2 Open Architecture

Open Architecture (OA) is a pattern of nonfunctional requirements (NFRs) that contribute to the ability to create, deploy and manage OA systems. In some domains, e.g. systems engineering, OA considerations would apply to both hardware and software components.

Open Architecture concepts have been around for years. In information technology it has been present at least since the 1981 introduction of the IBM PC hardware platform. In other domains, such as electrical and mechanical engineering it has often been seen, at least in key points where hardware or electrical systems from different vendors must cleanly, easily and safely interoperate. It can be argued that software engineering has been slower to adopt this approach because there are many more degrees of freedom to how components can interact than exist in electrical or mechanical systems. The increasing use of open standards in both COTS and Open Source software has increased the visibility of component-based, interchangeable software for complex systems and has shown OA to be viable.

The original IBM PC architecture is an example of a hardware open architecture; the World Wide Web is an example of a software open architecture. The idealized picture of such openness is simple interchangeability of components: “swap out, plug and play.” A more realistic view would expect a lowered barrier to entry and a greater velocity of technological improvement.

The oft-observed results of creating such an open architecture,

- rapid adoption of technology
- easier test and integration
- rapid improvement in technology capability and performance
- reduced lifecycle cost because of
 - increased competition
 - easier maintenance and upgrades
 - broader knowledge base
 - greater exploitation of reusability

have convinced many that including OA as a property of systems development and systems engineering can benefit their acquisition and development processes.

As is discussed below, the value derived from implementing OA technical principles depends on the existence of supporting business processes, including enterprise IT governance in addition to the fundamental processes around IT development and operations.

In short, OA specifies the business process and technical nonfunctional requirements that make it easier to assemble applications from replaceable functional components. As such, it applies to both the construction of a system (or system of systems) and the management of its lifecycle.

A lack of OA in a large system may result in the system not being capable of being upgraded. The complexity of building non-modular systems on a large scale historically results in very tightly integrated, but brittle systems that can only be scrapped if changes must be made.

1.3 OA Business and Technology Drivers

The fundamental drivers of OA are to reduce the total cost of ownership and the time to deliver IT systems. There are several common objectives that are identified:

- **Increase Reuse:** Increase the reuse of components across systems in all the lines of business.
- **Increase Flexibility:** Increase the flexibility of a system so it can be readily modified in response to business or technology changes.

- **Faster Time to Market:** Decrease the time required to develop, deploy, maintain and replace components and systems.
- **Reduce Costs:** Decrease the cost of development, deployment, maintenance and replacement of components and systems.
- **Leverage Competition:** Increase the opportunities for competition and innovation to both drive down costs and improve quality.
- **Improve Interoperability:** Create systems that are more readily interoperable with existing and future systems.
- **Reduce Risk:** Mitigate risks due to technology obsolescence or single source supply.

The principles and guidelines in this document are intended to support these business drivers, by identifying the key technical and business factors that facilitate the creation and deployment of OA systems. In the guidelines you will also find a general list of inhibitors and enablers of OA that are consistent with the OA Principles.

1.4 OA and SOA (Service Oriented Architecture)

The fundamental relationship between Open Architecture and Service Oriented Architecture can be summarized in five main points:

- OA is concerned with the quality of a component-based *system* or *system of systems*. SOA is concerned with the business functions that are provided *within and across* systems.
- OA features are likely to be found in a well-designed SOA.
- A system built to meet OA requirements is likely to facilitate the development of SOA services offered by that system at a later time.
- Many capabilities of a system are not exposed by a service interface and will never need to be. SOA is not relevant to their design or implementation; OA is always relevant to the design and implementation of the components of a system.
- Enterprise Architecture governance processes that support the conformance to OA principles have many objectives and activities that also support SOA enterprise practices.

1.4.1 OA Principles essential to SOA

1.4.1.1 Modularity

Service Oriented Architecture best practices rely on a modular software architecture that has carefully partitioned business and technical functions in a way that allows them to be independently accessed with minimal need to maintain state between client transactions.

It is possible for an SOA access layer to be placed over a non-modularized server application, but those services are likely to be less flexible, not at the right level of granularity and may invisibly maintain session state between transactions. Further, modularity is essential to horizontal and vertical scalability, which is also critical to SOA. SOA *must be scalable*, precisely because services are open to access to an unknown number of consumers.

1.4.1.2 Open Standards

The use of open standards most affects SOA where those standards have to do with service description, discovery, access, and nonfunctional requirements such as security and performance. Implementing these standards as a part of an OA system will make component services available to a broader SOA architecture.

1.4.1.3 Interoperability

Interoperability at a SOA level relies on well-defined interface syntax and semantics. Not only must an exposed interface capability be clear in what it does, but the relevant semantics of its data model must be available to the client so that errors don't arise due to the misinterpretation of data by applying the wrong context to its interpretation.

1.5 Document Scope

This document will address the technical principles of Open Architecture. It provides an Open Architecture reference model, a discussion of the business drivers behind OA and how the OA principles address the key driver of Total Cost of Ownership (TCO). Finally, it presents a set of organizational and technical practices that can operate as enablers or inhibitors to satisfying Open Architecture requirements.

The business principles will be addressed in a separate document. That document will address the business principles needed to acquire, develop and manage Open Architecture systems based on the technical principles described here. The document will differentiate between the principles needed in the private sector from those needed in the public sector. The principles in the public sector will be more extensive, because the acquisition of IT systems in the public sector is closely regulated, whereas there is much more flexibility in the private sector.

1.6 Related (future) Open Architecture White Papers

- Open Architecture Business Principles and Guidelines
- Open Architecture Metrics and Measurement

1.7 References

- [1] OASIS, *Reference Model for Service Oriented Architecture*, V1.0, 2 August 2006, <http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf>
- [2] Open Source Institute, *Open Standard Requirement for Software*, Draft 3, 2 September 2006, <http://opensource.org/osr/>
- [3] Open Source Institute, *The Open Source Definition*, V 1.9, no date, <http://opensource.org/docs/definition.php>
- [4] Open Systems Joint Task Force, *MOSA Program Manager's Guide*, Version 2.0, September, 2004. <http://www.acq.osd.mil/osjtf/pmguide.html>
- [5] Brown, Malveau, Hays & Mowbray, *AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*, 1998, John Wiley & Sons, Inc.
- [6] Department of the Navy, PEO-IWS 7, *Naval Open Architecture Contract Guidebook, Version 1.0*, July 2006, <https://acc.dau.mil/CommunityBrowser.aspx?id=105662> (Registration is not required).

- [7] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York, NY: 1990.
- [8] Open Systems Joint Task Force, *Open Systems Terms and Definitions*, <http://www.acq.osd.mil/ositf/termsdef.html>
- [9] Berkman Center for Internet & Society at Harvard Law School, *Roadmap for Open ICT Ecosystems, 2005*, <http://cyber.law.harvard.edu/epolicy/>

1.8 Acknowledgements

I would like to thank my colleagues at IBM, Timothy Fain, Lisa Yarbrough, Mark Cutler, Timothy Pavlick, Fred Mervine and others who helped me with advice and corrections on this paper. I also want to thank the significant contribution of the team of people working with the Navy, the Software Engineering Institute, The Johns Hopkins University and other who worked with on the Naval Open Architecture Initiative have made to my understanding of the issue. This paper is a distillation and extension of the work we did in understanding how to apply OA principles to the acquisition of the Navy's complex systems of systems.

2. Open Architecture Reference Model

This reference model provides an integrated overview of the key Open Architecture nonfunctional requirements and the business drivers that motivate them. It shows the relationships between the seven OA requirements. It also takes Total Cost of Ownership (TCO), shows how this business driver breaks down and how the nonfunctional requirements address the different elements of the driver. That relationship between the OA NFR and the business driver is meant to suggest that any design or process that satisfies the OA NFR is likely to address the need(s) reflected by the business driver.

2.1 Open Architecture Technical Nonfunctional Requirements

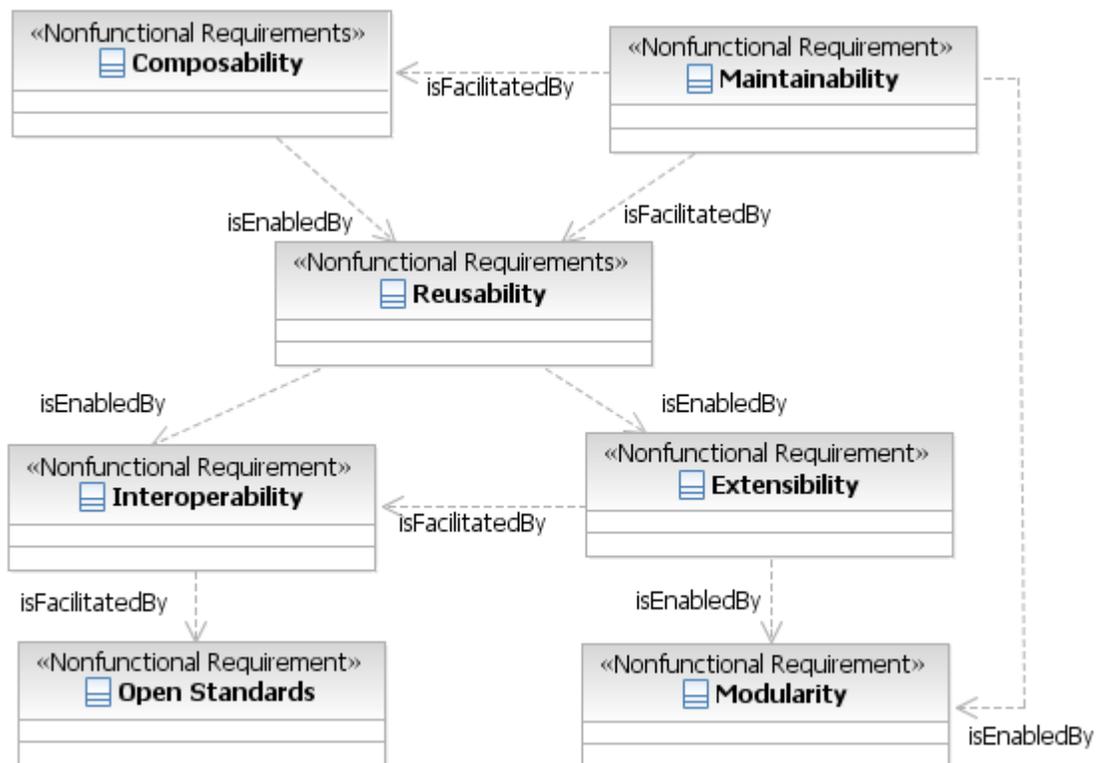


Figure 2-1: OA Technical Principles

There are strong relationships among the OA NFRs that are shown in Figure 2-1. The OA Technical nonfunctional requirements provide a pattern of architectural qualities that contribute to a globally open technical architecture. The sections below detail how the requirements are related to each other.

In general the figure reflects the fact that Open Standards and Modularity are the most fundamental requirements, while the others, to a greater or lesser extent, depend on the fundamental two and the other OA NFRs. There are two key relationships among these NFRs, *isEnabledBy* and *isFacilitatedBy*.

isEnabledBy: This relation implies that satisfaction of the NFR in an system design and implementation requires the satisfaction of the enabling NFR requirement in that system. Note that enablement is transitive: if A *isEnabledBy* B and B *isEnabledBy* C, then A *isEnabledBy* C.

isFacilitatedBy: This relation implies that satisfaction of the NFR is made easier by satisfaction of the facilitating requirement. Note that *isFacilitatedBy* can also be transitive, but because the relationship is not one of necessity, the impact of C in *facilitating* A is going to be less than that of B *facilitating* A, and less than that of C in *facilitating* B.

For example, Extensibility *isEnabledBy* Modularity, and *isFacilitatedBy* Interoperability. From these relations, one would expect that if Modularity is not present in a system, it will be very difficult for the system to be Extensible. If Interoperability is a property of a system, then it is likely to be easier to make it extensible, but its absence is not an inhibitor of Extensibility.

For each NFR, the implications of each relationship are detailed in the analysis below. The definitions of the principles in this section may be elaborated in the principles discussion in section 3.

2.1.1 Open Standards

2.1.1.1 Definition

Open Standards are "...publicly available documents that contain implementable specifications" (Wikipedia). The DoD defines it as "Standards that are widely used, consensus based, published and maintained by recognized industry standards organizations."²

The Open Source Institute has defined an Open Standard Requirement (OSR) that states the essential criteria of an Open Standard [2] for the purposes of Open Source software development. The requirement states:

To comply with the Open Standards Requirement, an open standard must satisfy the following Criteria:

1. **No Secrets:** The standard **MUST** include all details necessary for interoperable implementation.
2. **Availability:** The standard **MUST** be freely and publicly available (e.g., from a stable web site) under royalty-free terms.
3. **Patents:** All patents essential to implementation of the standard **MUST**:
 - be licensed under royalty-free terms for unrestricted use, or
 - be covered by a promise of non-assertion when practiced by open source software
4. **No Agreements:** There **MUST NOT** be any requirement for execution of a license agreement, NDA, grant, click-through, or any other form of paperwork, to deploy conforming implementations of the standard.
5. **No OSR-Incompatible Dependencies:** Implementation of the standard **MUST NOT** require any other technology that fails to meet the criteria of this Requirement.

² <http://www.acq.osd.mil/osjtf/termsdef.html>

It might be initially thought that criteria 3 and 4 don't apply to commercial software development or acquisition, since intellectual property protection and licensing are common practices. The criteria though, apply to the *ability to implement* the Open Standard not to any specific implementation of an Open Standard, unless that implementation is Open Source. Thus, a proprietary implementation of an Open Standard is possible. It can contain confidential IP, if the company so chooses. It can then be marketed as a proprietary implementation of one or more Open Standards, though if it violates the spirit of the OSR by having dependencies on components that violate the OSR; it is likely to be considered less than open by most practitioners.

2.1.1.2 Dependencies on other OA NFRs

None: The satisfaction of the Open Standards requirement does not depend on, nor is it facilitated by the satisfaction of any of the other OA NFRs. This is because the specification of an Open Standard lies outside of the system being designed and built. To satisfy the Open Standards requirement, relevant interfaces and protocols must be part of the system design and implementation.

2.1.2 Modularity

2.1.2.1 Definition

In software engineering, a *module* "...generally must be a component of a larger system, and operate within that system independently from the operations of the other components." (Wikipedia).

Modularity is a set of properties that support that independence of operations, which include

- Partitioning into discrete, scalable and self-contained units of functionality
- Well defined module interfaces, designed for ease of understanding (see reference [4]).

In Open Architecture, modularity is a measure of the clarity of the functional specification, and degree of independence from other system modules for its functionality. This applies to any particular bounded subsystem or components in the architecture. The specification does not need to be based on Open Standards. Using Open Standards interfaces does not inherently enhance the modularity of an artifact. Since it is, in a sense a measure of self-containment, it should not be expected to require the satisfaction of other NFRs in order to be satisfied by design and implementation.

2.1.2.2 Dependencies on other OA NFRs

None. Modularity is evaluated by the design and implementation of the module. Its focus is internal, in the sense that it does not directly make assumptions about its integration and runtime contexts. The requirement is only directly concerned with well-defined, bounded interfaces and behavior and minimal dependency on external interfaces.

2.1.3 Interoperability

2.1.3.1 Definition

Interoperability is the ability for systems and system elements running in separate process spaces to exchange data and information or request/provide the execution of a capability. The IEEE defines it as

... the ability of two or more systems or components to exchange information and to use the information that has been exchanged [7].

See 3.3 for a more detailed discussion.

2.1.3.2 Dependencies on other OA NFRs

Interoperability *isFacilitatedBy* Open Standards: Open Standards are, by definition [2], intended to interoperate in a distributed environment. Building a system using Open Standards makes the capabilities of that system a known quantity and it can expect other components that need the services specified by the standard will know how to interoperate with the system. It is not, however, necessary for a system, or its subsystems and components, to be based on Open Standards in order to be interoperable.

2.1.4 Extensibility

2.1.4.1 Definition

An extensible system has been designed with points of contact/integration (e.g. interfaces, ports, connectors, abstract classes) that allow future capabilities to be added to the component or system. To support extensibility, the internal implementation also has to be designed with sufficient internal quality and modularity of data and behavior that new capabilities do not introduce unintentional changes to existing data and behavior.

Extensibility refers to the ability to both add new capabilities to system components [module extensibility], and to add components and subsystems to existing systems [system extensibility].

2.1.4.2 Dependencies on other OA NFRs

Extensibility *isEnabledBy* Modularity: Systems that have tight coupling between components are very hard to extend because the web of dependence is very brittle. The effects of change are hard to predict and change is likely to cause malfunctions. Components that are not focused on well-defined business or technical capabilities have concomitantly “fuzzy” boundaries and it is unclear how they can be extended. A well-defined modular system does not suffer from these problems.

Extensibility *isFacilitatedBy* Interoperability: Systems that support Interoperability are hospitable to the inclusion of other, interoperable elements, and are, therefore more able to be extended. It is not necessary for a system to be interoperable in an external sense, if it has good modular design to enable its extensibility.

2.1.5 Reusability

2.1.5.1 Definition

Reusability is a property of an artifact that permits it to be used in multiple contexts to provide the similar capability in different contexts. Traditionally, in software engineering, reuse is seen at a code or module level. More recently, any lifecycle artifact in system design, development, training, implementation, configuration or maintenance artifact could be a candidate for reuse. Model Driven Architecture™ has made it more apparent, for example, that design artifacts are valuable reuse artifacts.

In an OA context, reusability does primarily refer to reusable system components, not as much to design and other artifacts that are part of the acquisition process. However, best practice suggests that all artifacts related to the design, construction and configuration of a component should be part of the deliverable reusable artifact.

See 3.5 for more discussion.

2.1.5.2 Dependencies on other OA NFRs

Reusability *isEnabledBy* Interoperability: Reusable components obviously need to be interoperable with other components with minimal adaptation. To the extent they are not their value as reusable components decreases.

Reusability *isEnabledBy* Extensibility: A reusable component is more easily integrated into a system that is easily extensible. The Reusable Asset Specification identifies points of extensibility as likely parts of an asset package. In some cases, a reusable asset should not be extensible, so extensibility is not a necessary feature, but principled points of variability in a reusable asset make it adaptable to more contexts.

Reusability *isEnabledBy* Modularity (transitively via Extensibility): Reuse depends upon both understandability of the artifact and its relative lack of dependency on other artifacts. A component that is tightly bound to other components in its runtime context is going to be hard to move to a different runtime context. Reuse in this scenario reduces down to duplicating the runtime context on another platform in order to reuse the component elsewhere. That's simple portability, not full reusability (see section 4.3). Good modular design provides the definition and minimal dependency that reuse needs.

2.1.6 Composability

2.1.6.1 Definition

Composability is a system design principle that deals with the inter-relationships of components. A highly composable system provides recombinant components that can be selected and assembled in various combinations to satisfy specific user requirements. (Wikipedia)

2.1.6.2 Dependencies on other OA NFRs

Composability *isEnabledBy* Reusability: Composability is an extreme version of reuse that does not require any modification or adaptation of the reusable component. Therefore, reusability in component and implementation is central to being able to achieving composability.

Composability *isEnabledBy* Interoperability (transitively through Reusability): Composable elements must be able to immediately collaborate with appropriate components in a new runtime context. This means that they must be interoperable and be placed in a runtime context that has a high degree of interoperability.

Composability *isEnabledBy* Extensibility (transitively through Reusability): Composability of system components requires, on the system level, that the insertion and modification of messages/operations is possible with minimal change to existing system components or system configuration. Further, logical changes to component interaction protocols should be possible without putting a component instance into an inconsistent state.

Composability *isEnabledBy* Modularity (transitively through Reusability and Extensibility): Systems with composable elements need to have very well-defined functional components whose capabilities are clear and well bounded. Further, the elements need to have a minimum number of dependencies so that they can be placed in different runtime contexts without failing. Good modular design is essential to providing those two properties.

2.1.7 Maintainability

2.1.7.1 Definition

Maintainability is “The ease with which maintenance of a functional unit can be performed in accordance with prescribed requirements.” (Wikipedia)

In OA the scope of maintainability is the portion of a component’s or system’s lifecycle after installation, including its end of life. Key to this lifecycle is updating the system to introduce new technology, changed business processes, etc.

2.1.7.2 Dependencies on other OA NFRs

Maintainability *isEnabledBy* Modularity: Well designed modular components make it possible to make changes to, replace parts of, and extend an existing system. Without the decoupling provided by modularity changes are much harder to incorporate, and over time take increasingly longer to create, test and install.

Maintainability *isFacilitatedBy* Composability: If a system uses recombinant components, it is easier to add, replace or remove components without system failure. In this sense, it is not essential, but it does help with the maintainability of a system.

Maintainability *isFacilitatedBy* Reusability: Reusability of components is primarily focused on the creation of a system with reusable components, so maintainability is not a direct concern. Nonetheless, the presence of reusable components, where they show modularity and interoperability are likely to be more easily maintained.

Maintainability *isEnabledBy* Interoperability (via Reusability *isEnabledBy* Interoperability): Components and systems that are developed with interoperability in mind, make it substantially easier to maintain the system through changes in technology, functionality and scale.

Maintainability *isFacilitatedBy* Open Standards (via Interoperability *isFacilitatedBy* Open Standards): Open Standards are likely to be supported by multiple implementations that should be interchangeable, and thus replaceable to perform patches, upgrades and replacements.

2.2 Open Architecture Drivers

The list of business and technical drivers can be found in section 1.3 above. This version of the whitepaper will only analyze the key driver Total Cost of Ownership. Other drivers will be addressed in later versions of this document.

2.2.1 Total Cost of Ownership

Total Cost of Ownership (TCO) is the total of all the expenses related to a specific technology. Figure 2-2 below shows the primary kinds of expenses that contribute to the TCO. This driver is generally considered the most critical to any organization. Many of the others drivers cited above can be seen as directly contributing to the TCO driver, including Increase Reuse, Leverage Competition and Faster Time to Market.

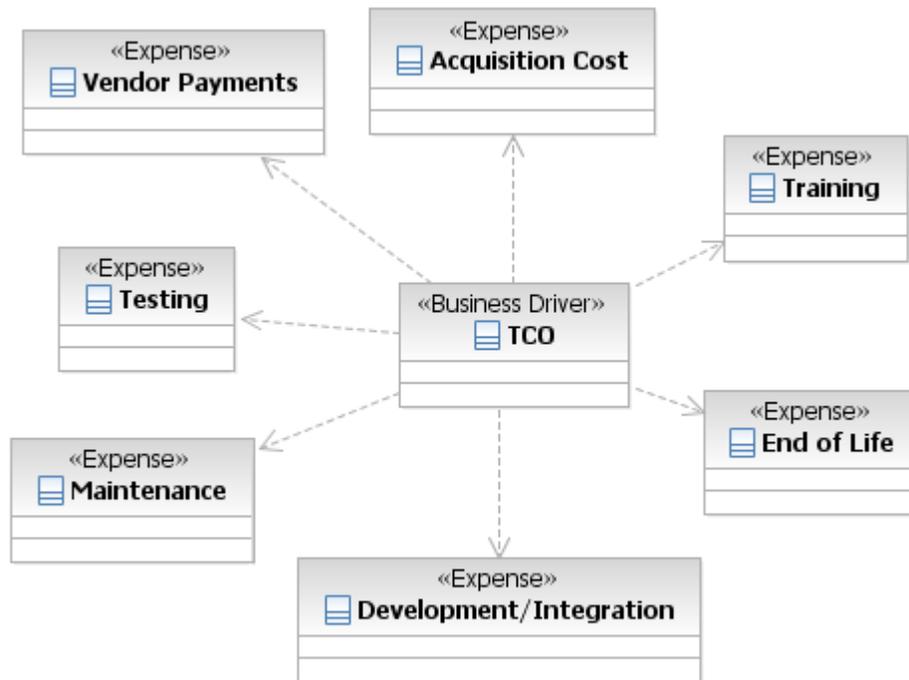


Figure 2-2: Total Cost of Ownership Expense Elements

Each of these expense elements are part of the total cost of ownership of a system. The section below briefly describes how each of the TCO elements can be addressed by one or more of the OA technical principles.

2.2.1.1 Acquisition

Acquisition expenses reflect the initial costs of purchasing the desired components or systems. It includes the product price and the labor of evaluating alternative products, due diligence in the ability of the vendor to deliver and support the product throughout the system lifetime.

- **Open Standards** contribute to decreased acquisition costs to the extent that there are multiple vendors that provide implementations of the Open Standard. This brings price and quality competition that will benefit the customer. This effect is greater if there is an Open Source implementation of the standard; even if a commercial vendor's product is selected, price and capability will be influenced by the existence of an Open Source alternative.

2.2.1.2 Vendor Payments

Vendor payments refers to any ongoing payments for the use of and support for a technology, where a 3rd party product requires ongoing payments (e.g. usage charge schemes) or technical support is contracted for or provided on an ad hoc, billed, basis.

- **Open Standards** contribute to decreased licensing and support costs due to the same competitive forces that reduced acquisition costs. Though this may not achieve the same cost reduction as a completely open source implementation, the availability of open source will still serve to drive vendor support costs downward.

2.2.1.3 Training

Training expenses refer to the cost of training relevant team members in the usage and technical details of the technology. This can include users, architects, developers, testers, maintenance technicians, system administrators, etc.

- **Open Standards** reduces training costs because it takes advantage of a population of experienced developers, well-developed instruction and open information. Additionally, it is easier to find skilled practitioners to hire, and if Open Standard components are used across an organization, a critical mass of technical skill can be developed within the organization.
- **Modularity** can simplify training costs because well-defined functional components can be studied in relative independence, and component behavior can be understood independent of system behavior.
- **Reusability** of components and systems means that as they are used in other systems and systems of systems, training received in one context can transfer to the new context, lowering the learning curve when moving from one system to the next.

2.2.1.4 Development/Integration

Development and integration expenses include all labor, tooling and secondary dependencies such as, technical infrastructure, middleware, technical support for developers, etc.

- **Open Standards** support development and integration in two ways:
 - Skilled developers and common knowledge of the technology.
 - Vendor implementations are supported by component and integration testing. This is not to say that it is perfect, but it is likely to be more robust than the internally developed equivalent.
 - Open standards implementations are often common enough that they do not have specific dependencies on tooling or middleware. In principle any dependency of this sort limits the openness of the component..
- **Modularity** makes it easier to develop system components in parallel. If the functional decomposition of the system is intuitively clear and straightforward it reduces the overall complexity of the system and makes it easier to develop and later integrate.
- **Extensibility** has two aspects:
 - **Component extensibility** requires additional effort (cost) in both design and implementation to create an extensible component. The value will come at a later point when it is easier to add modify the component without having to change components that are dependent on the changed component.
 - **System extensibility** is possible if subsystem and component modularity provides very loosely coupled capabilities, then additional components can be added with a minimum of follow-on changes to include the new capabilities.
- **Interoperability** facilitates the integration of new components into an existing system.
- **Reusability**, like extensibility, requires additional design costs, care in implementation and additional testing. It returns its investment, by some estimates, after the 2nd reuse (other estimates say after the 3rd or 4th reuse).
- **Composability**, like reusability, requires up front design work, but returns the investment in subsequent development.

2.2.1.5 Testing

Testing expenses include specialized testing tools, test preparation and management, unit, functional, integration, regression and user acceptance testing.

- **Open Standards** can reduce the extent of testing at a component unit level, if open source or vendor implementations are used. Integration testing may be reduced if other components of the system, especially 3rd party components, have been previously been designed and tested to be compatible with the relevant Open Standards. Over time, the use of OSR-conformant components can also reduce the cost of regression testing.
- **Modularity** best supports unit testing when well-defined interfaces encompass the full functionality of the component and testing can be designed before development is completed. Low dependency metrics in a system reduce the number of integration tests that are needed, because there are fewer points of variability at the system level.
- **Extensibility** can directly support testing by making it easy to instrument component and the system with no impact on the system functionality. Functional extensions to a component are also well-defined so extending unit and systems tests should be relatively straightforward and regression testing is unlikely to require extensive modification.
- **Interoperability** requires additional testing when components are created. However, OA returns the investment by facilitating integration in subsequent use of components thus leveraging prior testing efforts associated with component design and development.

2.2.1.6 Maintenance

Maintenance expenses encompass all costs associated with a technology after its initial integration and installation.

- **Maintainability** is, in large part, an emergent property of meeting many of the other OA NFRs. But it is still possible to create a system where maintainability is explicitly factored into support tooling, where maintenance use cases were specifically included in design and requirements analysis and documentation can be provided that directly addresses maintenance.
- **Open Standards** components can, if appropriately implemented, be replaced by other implementations of the same standard, thus simplifying component replacement. Version upgrades can be similarly straightforward.
- **Modularity** reduces cost in maintenance by making components easier to remove, replace and update because changes have limited impact, especially if there are no changes to the interfaces offered by the components.
- **Extensibility**
 - **Component extensibility** is an advantage when specific components need to be extended to provide additional capabilities. It does not directly facilitate bug level maintenance, but, where maintenance doesn't result in interface changes, there should be little impact.
 - **System extensibility** provides the greatest advantage in maintenance by making it easier to add new, or fix existing, capabilities to the system with minimal impact.
- **Interoperability** facilitates maintenance because components and subsystems created to work well in multiple runtime environments are likely to be readily modified and enhanced.

2.2.1.7 End of Life

- **Maintainability** decreases end of life costs by its attention to the ease with which modules and systems can be changed. Removal of a component, or system, in the context of a larger system or

system of systems is simply another type of change that should be accounted for. Maintainability is dependent on a number of other OA NFRs, which, transitively, should also be considered as affecting the cost of end of life expenses.

- **Modularity** decreases the difficulty of managing component end of life costs to the degree that there are a minimum number of remaining dependent components that need to be changed due to the removal.

3. OA Technical Principles

The core purpose of Open Architecture is to answer the following question affirmatively:

Is a qualified 3rd party able to replace a component of a system, based only on openly published and available technical and functional specifications of the component of that system?³

To answer this affirmatively, it is necessary that neither the technical architecture of a system, nor the organization's business practices, should impede the ability of a 3rd party to develop, sell, integrate and maintain a component.

The technical Architecture Principles of an Open Architecture are:

- Use Open Standards
- Use Modular Design
- Design for Interoperability
- Design for Extensibility
- Exploit Reusability
- Design for Composability
- Design for Maintainability

These principles are discussed in detail in the sections below.

3.1 Use Open Standards

An Open Standard is a comprehensive collection of APIs and functional specifications that are publicly available, specify the syntax, behavior, development and runtime dependencies, and identify the qualities of service required and provided.

Open Source implementations of an Open Standard should also adhere to the Open Source Definition [3], from the Open Source Institute. That may not be of direct concern here, though it is worth noting that an open source implementation of an Open Standard is a proof point for the viability of the standard and its conformance to the OSR.

Note: Open Standards are created by standards organizations and the result can suffer from the "Design by Committee" antipattern [5]. Implementation through Open Source mitigates that risk precisely because it is subject to the critical eye of a large community of skilled developers who attempt to implement the standard.

3.1.1 Use Open Standards Principle Statement

In an Open Architecture system, applicable Open Standards are mandatory specifications for all systems and components. This applies to middleware and 3rd party components as well as the system under development.

³ Claude Barron, personal communication.

Where no applicable Open Standard exists, components and systems (whether Commercial Off-the-Shelf (COTS) or Custom) should be documented and available, within the organization, in a manner consistent with Open Standards and the OSR.

3.1.2 Motivation for Open Standards Use

Proprietary or unpublished APIs lock system owners into a particular vendor's products and upgrade schedule. Because there is no ability for other vendors to provide competitive technology that would provide precisely the same service, system owners pay higher prices. In addition, because there is no competition and a captive market, the product capabilities do not benefit from competitive innovation. The vendor, in fact, has little incentive to invest in improvements to the product.

The objective of this principle is to create systems that have a maximum number of standard replaceable components during their lifetimes. In this way, cost reductions and quality improvements can be easily incorporated in relatively uncomplicated upgrades.

Use of Open Standards reduces the risks associated with integration and interoperability with new systems and components. Since industry generally sees Open Standards compliance as something customers want, new systems are often built to conform to the appropriate standards.

Additional value to an Open Standards APIs is the existence of a Community of Interest that deeply understands the APIs and has a number of designers and developers who are familiar with the standard. Rarely can the same be said of proprietary or undocumented APIs.

3.1.3 Implications of Using Open Standards

- In many cases the scope of an Open Standard API will not provide all the capabilities required to implement a system, especially if the overall system is central to the business. Avoid customizing or extending any components that conform to an Open Standard. Instead, implement the additional functionality in an extension component that explicitly depends upon the standardized component.
- Use of, and compliance with, Open Standards will have an initial learning curve in many organizations, especially in those IT organizations that have long built their own systems, or perceive their problem domain as being too constrained or unique.
- Conversely, the use of Open Standards will also make it likely that there are more skilled practitioners in the marketplace, and a larger community of interest will exist than can be found with many proprietary systems.
- The compliance with Open Standards will have to be assessed by an architecture review board that has enterprise-wide responsibility. If this practice is not carried out across an organization, then there will be less long term value to its use; this is because the organization will not develop a critical base of knowledge around using the standard, and the benefits of easier integration and interoperability will not be as available when cross-line-of-business integration is required.

3.2 Use Modular Design

Modular Design is “[a] design approach that adheres to four fundamental tenets of cohesiveness, encapsulation, self-containment, and high binding to design a system component as an independently operable unit subject to change.”[8]

Modularity is the degree to which a conceptually separate, well-defined unit of functionality

1. Is implemented as a component or set of cohesive components that provides a standard interface for the invocation of functionality and access to its state;
2. Has a minimum number of structural and runtime dependencies on other system components.

A conceptually separate, well-defined unit of functionality is best understood in the context of the common knowledge of a community of interest (COI). For example, a FundsTransfer web service would be a well-defined function within a financial COI, and an AuditLog component would be well-defined to a software architecture COI.

Open modular design rigorously specifies the public interfaces, data model, usage protocols, constraints, dependencies and qualities of service that are offered and required. This information is necessary for effective development using the system or component. Further, it should document, in comprehensive detail, all essential functional behavior.

Modular design should also take into consideration the lifecycle management of the system or component. The design should provide for easy reconfiguration, removal and replacement.

3.2.1 Use Modular Design Principle Statement

Modular design is required in all layers of the system, both for software and hardware.

3.2.2 Motivation for the use of Modular Design

Modular design makes it much easier to maintain, extend and upgrade systems, thereby reducing the total cost of ownership of a system. In conjunction with the Open Standards compliance it makes it much easier to take advantage of vendor competition in price and performance of standardized systems and components.

3.2.3 Implications of using Modular Design

- If modular design is not a common practice in an organization, then good design of a modular system will take significantly more time for design and development. Additional time will be required in developing:
 - Clear documentation of the modular components and system design. This is critical to the ability to reuse components and to maintain the system over time.
 - Data models and well-specified qualities of service to provide the most interoperability between components.
- If a module in a modular system is not provided by a third party, then the enterprise must establish standard ownership and lifecycle maintenance procedures. Modular components should not be modified ad hoc. There are two common approaches to this: owned management and organizational open source.
 - In owned management, one organization owns the component, its lifecycle management and deployment in the organization. This practice is probably best for an organization that does not treat its IT infrastructure as a strategic asset. All change requests (bug reports and additional features) go to the owning organization and are deployed in the normal change management processes, such as ITIL.
 - In organizational open source, OS collaboration practices are followed and all interested parties can contribute new features and fix bugs. The organization should study OS best practices before it commits itself to such an approach. In particular, it is necessary to have some aspects

of the OS process under Enterprise Architecture review to make sure that EA requirements are met. This will require a relatively active Enterprise Architecture team, which is sometimes a challenge if IT is not considered a strategic asset.

3.3 Design for Interoperability

Interoperability is a term, like reuse, that has several meanings when understanding a system. Interoperability is essentially the capability of a component or system to collaborate with other components or systems. The collaboration may involve message exchange, data exchange or method invocations. In an open architecture, interoperability is also based on commonly used open standards APIs and protocols.

For software systems there are several levels of interoperability that are used⁴. It is critical to understand, in any context, which meanings are intended. They are, in rough order of complexity:

- *Non-interference*: A minimal level of interoperability is where two or more applications can execute on the same physical platform, in different OS processes without interfering with each other's operations. They can share resources through the OS without limiting access or corrupting data.
- *Communications*: For any non-trivial degree of interoperability, the separate systems must communicate in some manner. To the extent that this interoperability is within the same physical computer, between computers with the same OS and Middleware, or between systems on different hardware, OS and Middleware provides a scale for how open it may truly be. The scope/range of the communications may also be a metric for how exposed the interoperation is.
- *Data Interoperation*: Increasing the degree of interoperability, one or more communicating applications can exchange data either through a shared data repository or by data requests through interprocess or remote communications channels and protocols. The semantics of the data need not be the same between the various systems. Each is only interested in the specific data for its own operation.
- *Semantic Interoperation*: At this level, the shared data have the same meaning for the interoperable systems. In principle any one of the systems could provide another with the same information and it would "mean" the same thing in all contexts.
- *Functional Interoperation*: At this level, shared semantics are essential to interoperating systems being able to bring their capabilities to bear on a common objective. A particular capability may only be available through one system, but separate systems know how to request that capability, with appropriate information, and understand the results of the execution of the capability.
- *Dynamic Interoperation*: At this level, a required capability can be found, when needed, from the available capabilities on the network. A trusted meta-data repository, based on a shared ontology, must exist to provide the information on the type of capabilities available, how to obtain them, how to interact with them, the data requirements and results and the available runtime qualities of service provided.

To some extent, reusability and interoperability are often mistakenly thought to overlap where interoperability enables capabilities to be shared. Interoperability can enhance reuse by making it easier to reuse parts to create a higher level solution, but so does modularity.

⁴ There are analogs and differences in dealing with hardware, affecting connections, common current, etc. I am not, however, currently that familiar with how interoperability issues could be analogously classified. That question will be addressed in a future version of the paper. Suggestions are welcome.

3.3.1 Design for Interoperability Principle Statement

Open Architecture systems and components must provide and enable the interoperability of systems, processes and data. OA interoperability is concerned with how the elements of a system can cleanly interoperate and how a system can interoperate with other systems.

3.3.2 Motivation for designing for Interoperability

Interoperability of application systems applies to the appropriate sharing of system resources and to the ability to provide and utilize capabilities and data between those systems. Full utilization of system capabilities in an enterprise leads to horizontal integration across the platforms, for which interoperability is a prerequisite.

Resource Sharing: Within a processing node and across a network, the increasing complexity of systems and the growing use of Service Oriented Architecture require that capabilities and data across the system are readily available to all applications across the system.

3.3.3 Implications of designing for Interoperability

- Common data models (a.k.a. standardized ontologies) will need to be established at an enterprise level, if not at a cross-organizational Community of Interest. Common data models, which include not only data structures, but also the semantics of the relations among data structures, are critical to effective interoperability. If a data point in a shared data structure does not have the same meaning in all applications that use it, there will be differences in how it is used, and ultimately how it may be updated. When considered in isolation this issue might seem to be one that is infrequently encountered, in fact, it is a major cause of the failure of system interoperability.
- Qualities of service may also need to be addressed as aspects of interoperability. If a client system requires a specific QoS for it to operate correctly, then the required QoS will be essential to the determination of interoperability.
- Information assurance (IA) is a critical assumption underlying correct interoperability. Without confidence in the data (and metadata) affecting executing behavior, the correctness of interoperation is in doubt. The need for IA does not apply only to data shared during runtime, but also to the provenance of the data, its security while at rest, its currency, and the security and reliability of its handling while in flight.

3.4 Design for Extensibility

Extensibility is the capability of a system to accept additional components with additional capabilities without extensive change to other parts of the system.

Although a modular, open standards compliant system may provide a plug and play capability; extensibility does not necessarily follow. Some components, though standards-based, may not be designed or implemented in a way that permits extension without extensive modification. This is especially true if substantial effort was made to optimize the performance of the component.

Extensibility is also a concern at the system level; is it possible for a system to readily accept and work with additional components with new capabilities without extensive change to other components or the system infrastructure.

Extensibility deeply depends upon the degree to which the collaboration of components in a system is decoupled from the implementation of the components. Where there is high cohesion between components in a system, new capabilities will be hard to add.

3.4.1 Design for Extensibility Principle Statement

Open Architecture components will be designed so that current capabilities can be improved and new capabilities added without substantial rework of the component or any components that depend on it. Open Architecture systems will be designed so that additional component capabilities and new components can be added to the system with little or no impact on existing components, capabilities and procedures. The system design should also minimize the impact of change on system management, maintenance and training.

3.4.2 Motivation for designing for Extensibility

Change is inevitable in an IT system. Simply by building a system, the users' perception of the problem domain is changed. They will see new aspects of the problem, or ways to improve it. In addition, external forces, and technology change, are always pushing change into the business practices and the system.

Systems that are not easily extended have been shown to take increasingly more time to modify as they are changed in response to new requirements, until the point is reached where new change requests arrive faster than old ones can be implemented.

3.4.3 Implications of designing for Extensibility

- It may be necessary to get stakeholders to express their vision of where they see the business going and how the system being built or enhanced will support that direction. Any system design represents a point in time; it addresses a current, and possibly near-future, set of concerns, but it is not necessarily expressed to the architects in terms of its role in the future. The stakeholders may not have considered it that way. If so, then the architects should encourage that conversation so that the scope of extensibility can be better understood. This will serve as a starting point for enabling, and justifying points of extensibility.
- Like all the design aspects of OA, expect additional time to be devoted to initial requirements and design phases of a project. Tightly integrated, non-extensible systems are straightforward to build, because they do not look beyond the requirements of the current problem. Designing in extensibility will take additional time.

3.5 Exploit Reusability

Reuse is an overloaded term that can cover everything from simple application portability, or cut and paste coding to J2EE components that can be deployed across J2EE application servers, to design element reuse. Essentially, any artifact related to the lifecycle of an engineered component may have some value to other, or subsequent, development or system management efforts. In most cases, though, the term is used with reference to the creation of components whose capabilities can be used in multiple systems that have different purposes.

In a strictly logical sense, this form of reuse is not a *necessary* feature of an Open Architecture system. A component could be based on modular, Open Standards designs with several vendors competing to provide it to customers without that component being usable in any other problem domain. Its capabilities may, for

example, be described in such a way that only specialists in a specific problem domain can understand them, or the nature of the problem is so unique it has no analog elsewhere.

Reusable components require a degree of abstraction from the specific problem domains in which they are used. They must capture the core behaviors and information common across most potential uses of the component (**commonalities**), as well as identify where the component is likely to have to be changed in order to work within a different problem context (**variability**).

Reuse, in a sense, is close to being an *emergent property* of the use of Open Standards, modular design and extensibility. Open Standards are usually based upon the agreement of a community of interest about the structure and behavior of a commonly used capability, or related set of capabilities; modular design drives the creation of a well-defined component with clear capabilities that has a minimum number of dependencies on other components, thereby freeing it to be used in different contexts; identifying and implementing points of variability in a reusable component is a basic aspect of extensibility. Components that meet these requirements are likely to be, in principle, reusable.

The question that remains is whether the functionality of a particular component is likely to be useful in other problem contexts. If not, then the additional design and documentation to package the asset into a reusable form is not needed.

The other artifacts of the system lifecycle should still be considered to be candidates for reuse. Requirements specifications, system architectures, application architectures, UML or SysML models, testing plans and scripts, documentation, maintenance manuals, project plans, etc. could all be used in other projects to decrease the amount of original work needed to create and manage a system.

3.5.1 Exploit Reusability Principle Statement

In each phase of the lifecycle of a system, maximum use of existing enterprise assets is required. Reuse is to be considered before extension, which in turn is to be considered before new design and development. The design and development of new components will take reusability into account and will submit reuse candidates to a review board.

3.5.2 Motivation for exploiting Reusability

Reuse of lifecycle assets can save substantial time, effort and increase quality if it is implemented in such a way that reusable assets can be understood as applying to a particular problem, within specific activities of the system lifecycle. It is crucial that assets be re-used at the right level of granularity; too much, or too little detail in an asset, relative to the activity involved can create more work by the need to transform the information into the right level to assist with design, implementation, or whatever activity is underway.

3.5.3 Implications of exploiting Reusability

- Design and implementation of reusable components requires several times more effort to develop, test and deploy. In addition to correctly abstracting the reusable data model and functionality (what is common to all uses of the component), it is necessary to identify what parts of the component will have to be customized in each new use (points of variability).
- If reuse is to be of substantial value to an enterprise, it will require management. The enterprise will have to establish some method by which reusable components are proposed, validated and made available; it will also need to establish methods to assure that new projects make the most use of existing reusable assets. This can present an organization with challenges in how to share the costs, ownership and

cross-organizational responsibilities. See the Open Source implications sections for two approaches to managing reusable assets.

- Reusable components will require extremely rigorous testing. In normal development, the use cases that specify system behavior are used to generate test cases. Testing outside the scope of usage in this context is not needed. However, a reusable component is “intended” to work in use cases not yet anticipated, meaning that some of the collaborations message sequences between the reusable component and other components will not have been anticipated. Exhaustive testing for the permutations on the use of the API calls may not be practical, but testing must more extensively exercise the behavior of the components. In particular, the testing needs to make sure that the test suite does succeed in executing *all* of the component code, even if it doesn’t invoke all possible call sequences. This should be considered a minimum acceptable standard for a reusable component testing.
- Use of the Reusable Asset Specification, an OMG standard, should be taken as a corporate standard for the packaging and deployment of reusable assets. This is because tooling for both creating reusable assets and storing them in searchable repositories are being developed and improved.
- Where component reuse is a corporate strategy, it is not always clear to the team that owns an asset the types of reuse it may be subject to. Often, in practice, it is necessary for potential component adopters to work with the owning team to specify the needs that they have for the component and then work with the owning team to assure correct development.
- Risk: without active management and processes that encourage reuse, asset repositories will simply become dumping grounds for large numbers of files.

3.6 Design for Composability

Composability, in general, is about how well the parts of the system go together. Composable components permit the creation of new composite services by orchestrating the functions exposed by the components. Such composability requires a high degree of modularity and well-defined syntactic and semantic interoperability of the components.

If you take away the requirement that it is possible to orchestrate a collection of components to implement a new process without changing the implementation, then you effectively have the requirement of interoperability.

So, like reusability, composability is an emergent property of the realization of other OA nonfunctional requirements. Good modular design is useful in providing a well-defined and well-understood capability that can be easily understood in many different use cases; depending on its deployment it may also be important that it has few dependencies on other components. Interoperability is critical to composability because composable components must work from a common data model if they are to be orchestrated to work together without change. This interoperability is most powerful if it is based on Open Standards because they tend to have been well-tested and understood by a large COI.

3.6.1 Design for Composability Principle Statement

System components that implement business level capabilities must be designed in such a way that it is possible to orchestrate their atomic behaviors into different sequences of action to implement multiple processes without having to change their implementation.

3.6.2 Motivation for designing for Composability

Business processes often change more frequently than many of the actual business capabilities that are implemented in IT systems. It is desirable to be able to take the implemented capabilities and re-order/reorganize them so that they provide new capabilities.

In principle, this is no different from what a programmer does in reordering the instructions of a programming language to perform different types of functions; this approach simply conceptualizes business capabilities as the “instructions” of the business’ action language.

3.6.3 Implications of designing for Composability

- Composability requires a business operation level specification of the capabilities of the IT infrastructure. Actions and information must be presented in terms that are very well-mapped to the vocabulary of the business. Being able to reorganize the way the capabilities work together requires they correspond very closely to how the business process experts understand the business to operate. Consequently, business and IT must communicate very clearly, and this communication will have had to be established over an extensive period of working closely together.
- If these capabilities are well-enough defined to map naturally to the operational concept of the business, then it is very likely they are also at a level of granularity that readily enables the use of SOA architectural approach.

3.7 Design for Maintainability

Maintainability is the degree to which a system’s

- capabilities can be sustained
- configurations can be changed,
- repairs can be applied and
- replacements can be installed.

In part, maintainability depends upon good modularity of components: when there are well-defined interfaces and few dependencies to a component needing replacement, it minimizes concomitant changes to the rest of the system.

Maintainability is also very much a feature of an entire system, or any set of integrated components that may make up a modular subsystem. It is possible to build a system with high quality modular components and still make it nearly impossible to physically access many of the components without taking the entire system apart. Thus, maintainability should also be a requirement on the overall architecture as it combines the components into an integrated system.

Maintainability also depends on well-documented procedures for reconfiguring, repairing and replacing a component in each subsystem and system

Although it may not seem relevant, a system with components that have well-defined, discrete functions is also easier to maintain because the purpose of each function is readily understood. This makes the assessment of how well it is operating a much more transparent effort. Technicians and users new to a system do not face as high a learning curve to understanding the system and its components. A system whose purpose and functions are clear will always be easier to manage and maintain.

3.7.1 Design for Maintainability Principle Statement

Open Architecture systems, subsystems and components must be designed and implemented in such a way as to minimize the effort needed to repair and manage them throughout their lifecycle.

3.7.2 Motivation for designing for Maintainability

Only about 20% of the lifecycle cost is in the development or acquisition. Of the rest, infrastructure and licenses are relatively fixed. Maintenance is the portion of the lifecycle cost that can be most reduced by upfront design

If there are logical, physical, legal or financial barriers to being able to maintain the parts of the system, then it isn't open -- even if the system is made up of modular components with Open Standards APIs.

These barriers will increase the Total Cost of Ownership of the system, they will require more time in deployment, maintenance and replacement of the system. This is of particular concern where the systems are mission- or safety-critical, such as automated manufacturing or aircraft controls.

3.7.3 Implications of designing for Maintainability

- Maintainable design requires additional time. Design of maintainable systems, like good modular design, will take additional thought.
- Determination of maintainability requires a clear understanding of the conditions of use of the component(s) and the system(s). Maintainability in a controlled environment like a server installation will be considerably different from maintainability in a hostile environment, such as the battlefield, or in extreme climates.
- Maintainability will need to be measured. Because maintainability, in many cases, must account for human activity in the maintenance process, measurement and analysis should include the time, effort and feasibility of maintenance by trained (and possibly untrained) technical workers. Metrics may include;
 - Tests of removing and replacing components.
 - How much time is required?
 - How long does it take an appropriate technician unfamiliar with the component to learn to replace the component with the provided documentation?
 - How long will it take an *untrained* person to replace the component If the environment requires it (e.g. in military systems, where there is a real risk of losing skilled personnel) and the component(s) are mission- or safety-critical).
 - For hardware components, physical accessibility.
- For some components, if there are reliable MTBF (mean time between failures) numbers, those can be factored into the overall design. For example, for low, or unknown, MTBF components effort must be made to make those components highly accessible and replaceable. For very high MTBF measures, the component(s) could be less easily replaced, *if other design constraints make it advisable*.
 - Where MTBF metrics are not known, they should be collected.
- Where possible, ongoing component health metrics should be collected, so that component replacement requirements can be projected.

4. Related Nonfunctional Requirements

Open Architecture NFRs facilitate the satisfaction of other NFRs, or are roughly synonymous with them. This section describes some of those implications.

4.1 Scalability

A scalable system is capable of increasing its capacity. Capacity can be defined as number of concurrent users, increasing throughput via faster processing, more processors, more threads per server/partition, or across more servers without serious degradation in overall performance for any individual transaction or execution thread. There are often referred to as Scale-Up or Scale-Out.

The modular design and interoperability principles are essential to the potential scalability of a system. The enablers and inhibitors associated with them also affect whether a system can readily scale vertically or horizontally.

4.2 Replaceability

Replaceability is a synonym for “plug and play” and modularity, interoperability and reusability requirements all apply to this NFR.

4.3 Portability

Portability is a special case of reusability, usually applied to being able to move an application from one hardware/OS platform to another. In some cases it can result in the application dependencies driving the decisions on what hardware and middleware are needed. Reuse generally aims to be much more platform independent. If it is locked to a platform, it is not open, and is more at risk of becoming obsolete when that platform is no longer available.

4.4 Supportability

Supportability in this context can be considered an aspect of the maintainability principle. The ease with which a system is understood by IT support and maintenance team, and the easier it is to repair, replace or upgrade, are the essential concerns of both supportability and maintainability.

4.5 Affordability

Affordability is the consideration of TCO within an enterprise’s budget planning. “Does the TCO of the system fit within that part of the IT budget? One of the key purposes of OA is to reduce TCO, so in that sense it can be said to intend to increase affordability.

4.6 Information Assurance (IA)

Information Assurance is not directly related to OA. But it is an essential enabler of interoperability. If the data and data semantics shared between processes are not trustable, then interoperability may work on a syntactic level, but on the results of collaboration are no longer reliable.

5. Open Architecture Guidelines

Open Architecture Guidelines identify practices that should be followed so that the OA business objectives can be realized. This section can be considered an initial listing of Best and Worst Practices for implementing technical OA.

5.1 Open Architecture Criteria

Each architect of a system should ask whether the OA nonfunctional requirements are relevant to a particular system being built, or to particular subsystems or components of the system. The following criteria, though not exhaustive, can be used to identify most cases.

5.1.1 *OA is needed if*

- The system or component is part of a mission-critical system that has a managed lifecycle. It is critical to the TCO of such a system that it be easily, funded, sponsored, maintained and extended by several generations of LOB executives and technicians.
- The system or component collaborates with other systems in a distributed computing environment. This includes client/server systems, Web-enabled applications, Service Oriented Architectures, federated middleware environments, etc.
- The system or component will be used in multiple runtime contexts. Reusable assets fall into this category

5.1.2 *OA is not needed if*

- The system or component is a prototype with a very specific scope and a very short lifecycle (effectively, it is a “one-off”). For example, developing a new algorithm for DNA analysis using new types of hardware.
 - **RISK:** Organizations frequently approve “prototypes,” claiming they realize it is not being built for production, and then later, using market timing as an excuse, drive them to market before redesign.
- The system does not interact with anything else.
 - This begs the question “Why are we doing this?” when today’s default IT ecosystem uses distributed computing, remote access, and the World Wide Web.
- The component is an informal script (e.g. UNIX shell, makefile, Perl, Ant) that is procedurally oriented and is used ad hoc by specialists who maintain it directly.
 - **Note:** This does not remove the responsibility to make the script maintainable and reusable, since there may be many developers using it, and it may be a required part of a reusable asset.
- The system or component will only be used by a small number of people, in the same organization and the designers and implementers will always be available for maintaining it. Further, the organization has no intention of ever using the system for any other purpose.
 - **RISK:** All of these are dubious assumptions.

- If it is easier to re-implement the system than to update it *and* the system does not interoperate with anything else. For example, some very simple Visual Basic applications can be handled this way, because they are effectively just another form of informal scripts.

5.1.3 “Degrees of Openness”

Since most systems are a combination of existing and new components and systems, it is not feasible to instantly become OA conformant. It may be necessary to identify, for business purposes, which OA requirements will be both critical and cost-effective to implement in the next iteration of the system. Those components and systems will then be considered candidate for redesign along more OA lines.

For example, an organization may have no interest in fielding a system of composable capabilities, in which case it won't be necessary to worry about composability. But if you are creating an open distributed processing system, then composability requirements could be of concern. In either case, modularity, reusability and interoperability can still affect the speed with which the system can be built and the ease with which it can be maintained.

If there is an existing standalone application that needs to become “SOA-enabled,” it is at least necessary to create an interface level that provides a degree of interoperability. This may make the application somewhat open, but it won't have all the advantages of OA.

- **RISK:** Such “silo” applications are rarely built with reuse and modularity in mind. More critically, their data models are likely to be application-specific, and therefore not consistent with an enterprise data model. This will be a serious limiting factor. Further, exposing their services to network-based clients may stress the scalability of the application, which may have been the “private” tool of a small number of workers.

It is also possible to view openness by the “scope” or “range” of an open architecture. An architecture may be open at a LOB level, where the data models and processes are standardized on a LOB data model and interoperability standards, but it may not be able to interoperate on an enterprise level. At the next level, it may be open across the enterprise, but not outside it. Further up, it could be interoperable across a Community of Interest, adhering, for example to industry-defined common data models.

5.2 About OA Enablers and Inhibitors

For each of the architectural principles, there are technical and business techniques and practices that can either help or hinder the development of a system that is consistent with the principle. This section takes each of the technical principles and presents a representative set of enablers and inhibitors.

This section does not claim to be a complete collection of the practices that enable or inhibit Open Architecture, but they do represent a number of the key ones.

Note that, obviously, an absence of an enabler, or a condition that is the opposite of an enabler should also be considered an inhibitor.

5.3 Organizational Practices

Although these technical architectural principles can be applied just to the design and implementation of a single system, Open Architecture practices have the greatest impact when applied to all system and software engineering within an enterprise. This means that there are business practices that support or inhibit the

effectiveness of use of technical OA principles in the organization. Managing the application of the OA principles then becomes an issue of IT governance, and should be part of the Enterprise Architecture portfolio.

5.3.1 Enablers

- **Managed Enterprise Architecture:** A number of the principles provide more business value if they operate within a context where there is an enterprise approach to the IT Architecture, Enterprise Architecture management and cross-organizational agreements regarding data, interoperation, reusable assets, etc.
- **Enterprise Architecture:** An enterprise view of organizational capabilities and how they are organized and deployed provides a sound basis for identifying redundant functionality, over- and under-investment of effort relative to value to the business and other structural and process problems that will make an organization less flexible and less focused. Since one of the OA business drivers is to increase flexibility (See 1.3 above) a clear picture of what the organization does is essential, otherwise the effects of any changes will be unpredictable and flexibility will be very hard to achieve. The functional architecture should provide at least the following:
 - A single instance of each Business Capability. It can be owned by a line of business, but it must be available to all business processes.
 - Precise and complete identification of each Capability's services provided and services required.
 - Identifies all Processes and the Capabilities used. Special attention should be paid to those capabilities used in multiple processes.
 - Identifies the Services specifically required by the processes, differentiating between those that require IT support or full implementation and those that don't.
- **Enterprise Standard Ontology:** Lines of business, the enterprise and IT tend to internally communicate with specialized vocabularies. Even within IT, it is not uncommon for different practitioners to have different concepts around the same technical term. For IT services to work across an organization, the full semantics of essential business information *must be standardized*. If different LOBs define "Customer" with different properties and different relationships to other business data, e.g. "Account", their applications will treat Customer differently and make changes that are likely to be mutually incompatible. Both the properties of "Customer" and the meaning and names of all its relationships in the business must be agreed to across the organization. One approach is to create a single organizational data and metadata repository for use by all systems. To achieve this on an enterprise level, organizations often create
 - Communities of Interest that standardize their internal ontologies and collaborate with other Communities to harmonize them across the boundaries. They may also choose to use, or customize, industry standards as the basis of their internal standard.
 - Establish ontology registries, libraries or metadata repositories that serve as a single point of publication and reference for these internal standards.
- **Layered Architecture Style:** Layered application architectures permit a separation of concerns in the development of components for the system. It also facilitates reuse of components. Components in a layer operate at a specific level of abstraction that makes it easier to develop and maintain. One example of a common implementation is:
 - **Customer Channels**, e.g. Web Services, Servlets, Java Message Beans, RPC, etc., which manage the protocols for requests from clients.

- **Process Choreography**, which composes the services into well-defined processes.
- **Business Services**, which are well-defined business services/operations, e.g. Funds Transfer, that coordinate the actions of Business Objects. In the case of a Funds Transfer it might coordinate the actions of multiple Account objects, a Customer object and enterprise services like Account Validation, Authentication and Authorization, etc.
- **Business Objects**, which are the “things” the business works with, and which provide fine grained control over state and actions. They also interact with other business objects and with access services that send and receive messages, access data, drive devices, etc.
- **Connection (or Access) Services**, which provides a technology independent interface to access data and services from whatever the current source is. JCA (Java Connection Architecture) is an example of such a layer.
- **Connectors**, which are technology specific access connectors that are used by the Connection Services layer to reach data and service resources, both within the enterprise and outside it.

One important feature of a layered architecture is the constraint that components in one layer not access services from a layer above. Violation of that constraint severely limits the reusability, flexibility and clean modularity of the component and the architecture in general.

- **Enterprise Service Bus:** The use of a message bus to route, transform and mediate messages between applications, including capabilities such as publish/subscribe and complex event processing is a very powerful method for decoupling consumers and producers in a networked environment. Combined with an Enterprise Standard Ontology, it facilitates the development of flexible, modular and reusable systems.
- **Identify and Manage Key Interfaces:** Key Interfaces are those that:
 - Are exposed to network-based access, such as web services, or RMI/IIOP, etc;
 - Are publicly accessible, within a common runtime.
 - Are stable and central to the functional specification of a system or component.
 - Have many components and systems that depend upon them.

Key interfaces should be identified early in the design and development phase and managed using change control throughout their lifecycle. Changes to key interfaces must be carefully assessed, the extent of the impact reviewed and the transition from the current version carefully managed.

- **Capability Specification of Requirements:** When requirements are described as capabilities (e.g. what needs to happen when), the qualitative required behavior is the focus of discussion. This approach to analysis appropriately separates what is needed from how it will be performed. Capability specification can state *how well* the behavior must perform (e.g. a funds transfer request must complete within 3 seconds.) These are the qualities of service (QoS) nonfunctional requirements that apply to the capability.
- **Model-centric Architecture:** The use of Model-Driven techniques, whether called Model-Driven Architecture™, Model-Driven System Development (MDSD), etc. is the foundation of a model-centric approach to system specification and governance. Model-centric architecture retains a semantically consistent representation of a system, or system of systems, and provides stakeholder-specific views of the model that allow the stakeholder assess the architecture from the perspective of their specific concerns. This approach supports an open architecture because it makes requirements and design artifacts more directly accessible, understandable and reusable. Models can be shared without loss of information, and the consistent semantics means that users will understand the models similarly.

5.3.2 Inhibitors

- **LOB-centric IT departments:** The more autonomous a line of business is, especially with regard to the ownership and management of its IT assets, the less likely its systems will interoperate well when any attempt is made to develop an enterprise strategy and management. This is especially true when attempting to create an enterprise data model: data ownership, including the definition of data is often difficult for a LOB to cede control over.
- **LOB-centric Procurement Practices:** Separately negotiated contracts and long-established relationships between LOB executives and vendors provide strong motivation to maintain autonomy from any enterprise integration. For example, some cost savings deals may have created penalty clauses if contracts are prematurely terminated. Integration across the many resulting technologies and architectures will present massive technical and cost challenges.
- **Vendor-Driven Technology Strategy:** Vendor product lock-in is inherently contrary to the goals of Open Architecture.
- **Technology-Driven IT:** The drivers of Open Architecture are business drivers. IT departments that are not driven directly by the business' strategic and tactical needs will not be able to deliver the appropriate architectures and services. The last thing you want to hear from IT is: "Hey, we built this cool tool for you! See if you can make some money with it!"
- **IT Viewed as a Cost Center:** The level of organizational commitment needed to get the most value from an open architecture requires that IT be well integrated into the strategic planning of the organization. If IT is seen as the builders of tools to speed up work on individual problems, or the generators of paper reports, then there will be little to gain from an Open Architecture approach
- **Obsessive local optimization:** Organizations that have a history of developing high performance, single-task systems develop common engineering practices that place a very high value on optimizing all aspects of a system. The problem that arises is that a high performance system builds in a very large number of design decisions that are tuned to the specified problem only: data models are specific only to the problem, code is optimized to take advantage of relationships between objects in the problem domain (where the object may have other relationships and functions in other contexts of the organization). Interoperability, reusability, standards compliance, and extensibility are generally sacrificed to achieve optimal local performance. As a result, enterprise level performance will end up being sub-optimal.
 - **Note:** There are still some cases where current COTS technology is not capable of the necessary performance, but the number of those cases is decreasing almost daily. Careful thought needs to be given in these cases to make sure that the optimization is local to the needed capability and that the larger system is still conformant with OA principles.
- **Common use of stateful transactions:** Stateful transactions are effectively a tight runtime binding between client and server. The need, on the server side, to maintain state in memory across transactions constrains the horizontal scalability of the system in two ways:
 - **The number of concurrent users per server.** If state must be maintained in memory between transactions, then at some point, there is no longer any free memory.
 - **The system's ability to load balance.** If all of a client's transactions are bound to a specific server, then the system cannot as easily balance the workload across servers.

There are technological approaches to this problem, such as saving and restoring session state to and from common network storage, but the technology and effort costs are sometimes not worth the return, and there will be some loss of performance which may also be a concern.

5.4 Open Standards

5.4.1 Enablers

- **Open Source implementation of an Open Standard:** If there exists an Open Source implementation of an Open Standard, then it is likely that the standard is consistent with the Open Standard Requirement (OSR) [2], and other implementations, whether Open Source or provided by vendors, are more likely to be compliant with the OSR because of competitive pressure.

5.4.2 Inhibitors

- **Single Vendor Implementation of an Open Standard:** If there is only a single implementation of an open standard, it may be that the “standard” was pushed through the standard setting process by the single vendor. There is some risk here that the implementation does not conform to the OSR. On the other hand, the openness of the standard exposes the specification to change from outside of the single vendor by a COI, and other vendors may eventually come into the market. The risk is less than with a proprietary system, but not as good as having an existing Open Source implementation.
- **Violations of the Open Standard Requirement:** The OSR is a concise description of the types of dependencies that limit the value of using Open Standards implementations. (See 3.1 above.)
- **Standardizing on Platforms Instead of Interfaces:** Standards should be based on specifications, not on implementations. For example, it is useful to standardize on J2EE but not on a specific vendor’s implementation of J2EE, e.g. BEA, because systems will be implemented to the platform and invariably develop to the non-standard extensions.
- **Absence of Open Source Implementation:** If there is no viable open source implementation of a standard after a certain number of years, be cautious of using implementations (a variation of the Single Vendor problem) because the standard may not be internally coherent. There may be other reasons, such as a very narrow problem domain, but more careful analysis should be pursued.

5.5 Modular Design

5.5.1 Enablers

- **Object Oriented Analysis, Design and Programming:** Modular design is more easily expressed because of the separation between interface and implementation that is essential to OO practices.
- **Simple Design:** A good OO design should be simple, and is almost demonstrably best if, when it is shown to a domain expert, the response is: “Well, of course. That’s obvious.” A simple design that captures the essential abstractions and presents the domain in a clearly understandable set of concepts and relationships will be much easier to use, maintain and extend.
- **Open Standards APIs:** Standards provided by international standards organizations, in general, have been extensively analyzed for comprehensiveness, clarity and completeness. As such, they usually provide a good modular design for a problem domain or technology infrastructure. Further, products implementing the standard remove both risk of errors in internal design and substantial reduction in internal development expense. The cost of purchase and licensing may offset the TCO if the same

capability is developed internally. As an open standard, costs are likely to drop as competition between vendors, or an open source implementation is provided.

5.5.2 Inhibitors

- **Structured Design Techniques and Languages:** These languages support a more function-oriented approach to development, and the separation of interface from implementation is not an easy thing to achieve. Any resulting design and implementation that tries to impose an interface/implementation using a structured language is likely to make the design and code artifacts difficult to understand and maintain. Additionally, after any original team leaves, the maintenance developers are likely to slowly modify the code back into a form more common to the particular language being used.
- **Complex Design:** An OO design that has many different types with a large number of relationships is often a bad design. It has not captured the abstractions in the problem domain, but is likely to be an OO model of the existing implementation. A complex design will be very difficult to maintain, will tend to be brittle, may not perform well and is unlikely to scale.
- **Legacy system migrated directly to a distributed system:** When an existing system is ported to a distributed one without substantial redesign it will increase the complexity of the enterprise system, be very hard to interoperate with and will inhibit the partitioning of the overall system into well-defined modules. This is because it likely represents a LOB problem-specific perspective, not an enterprise process perspective. It is likely to make very specific assumptions about location, address space, and access to local file systems and other local resources [5].

5.6 Maintainability

5.6.1 Enablers

- **Modular Design Enablers**
- **Modular design with well-defined, stable interfaces:** If different releases of the same component(s) have a clear and stable interface over time it will reduce training needed to understand each new release, and will build a broader common knowledge base across the organization.
- **Loose coupling between components:** Minimizing the impact of change greatly speeds up the time it takes to replace, repair or reconfigure a component.
- **Clear, comprehensive but concise documentation:** Maintenance is usually a task given to new technicians, meaning that they will be facing a learning curve for each component for which they are responsible.
- **Maintainability Use Cases and Testing:** Maintenance use cases should be part of the analysis of the component and should be used to create testing scripts for the configuration, removal and replacement of the component.
- **Open Standards compliance:** Expertise in Open Standards systems exists outside an organization. This makes it easier to find skilled technical workers to work on a system.

5.6.2 Inhibitors

- **Modular Design Inhibitors**

- **Frequent changes to component and system interfaces:** Frequent changes to key interfaces reflect an unstable design or a constantly changing set of requirements. The system has either been poorly designed, or is not ready for production.
- **Functional Decomposition:** This approach to analysis directs design away from abstractions toward solutions tightly coupled to the problem domain and heavily optimized, but difficult to change because there is very high cohesion across the parts of the system.

5.7 Interoperability

5.7.1 Enablers

- **Maintainability Enablers**
- **Detailed Specifications of Internal Design Elements:** Where it is necessary to develop key interfaces internally, they should be designed and documented on a level of quality equivalent to that of Open Standards. This will help develop a common base of knowledge across an organization.
- **Accessible Standardized Enterprise Metadata Repository:** Syntactic interoperability between applications and systems is easy to establish and maintain: systems will not collaborate if the call and return syntax are not correct. It is much harder to establish an enterprise level common data model and the metadata that describe it, yet IT system collaboration across traditional application and Line of Business silos will not succeed until all systems understand the business' data model (including deep semantics) in exactly the same way.
- **Standardized Data Models, and Metadata established by COI:** Enterprise-level data models and metadata are best when they reflect the expertise of a Community of Interest. They should not be created by enthusiastic amateurs, however clever. They need to reflect experience as well as domain knowledge.
- **Data are available (posted) when created:** Non-trivial interoperable systems often exist to provide a current view of the state of an organization, or provide valuable situational data to workers. Data and information need to be available to the organization as soon as they are valid. For example, a securities trader needs to know the most recent trade(s) of a security as soon as they occur, not after batch process and the end of the day. Posting data when it's created opens many more opportunities to use the information available to the organization in new ways.
- **Web Services Discovery and Invocation Capabilities:** Web Services is a well-known protocol for interaction between applications across the network. Exposing services by way of a registry and repository provides a well-specified interface and programming model that lowers the barrier to implementing interoperable systems.
- **Enterprise-wide Information Assurance (IA) Practices:** Reliable operation of interoperable and interacting systems requires confidence that data, in flight, under transformation and at rest is not subject to unauthorized change. Note that this is independent of information security. IA applies to both clear and encrypted information.
- **Consumer / Supplier decoupling through message-, or event-based service bus:** Runtime decoupling of consumer and supplier means that client and suppliers only need to share a data model in order to interoperate, other technical considerations like APIs, runtime platforms, interlanguage communications, competition for resources (disk, memory, CPU, ports, etc.) are less critical.

5.7.2 Inhibitors

- **Maintainability inhibitors**
- **Proprietary or unpublished APIs:** Non-open APIs present a barrier to entry to interoperability with a system. They can present legal constraints, or difficulty in finding skilled technicians, and frequently present a serious challenge to understanding how the implementation really works.
- **Point to point connectivity:** Systems that require direct point to point connections between interacting applications or systems do not scale, and require significant time to integrate new capabilities.
- **Application data models renamed as Enterprise Data Models:** Application data models are designed with a single problem domain in mind. For most, the best practice was to optimize the data model to include only the data needed for the application to operate correctly. This local optimization intentionally avoids addressing enterprise level information needs, and therefore is extremely unlikely to translate well at the enterprise level.
 - **Calling the union of application data models an Enterprise data model:** Historically, systems and applications have been built to support difficult problems faced by a specific line of business. There are relationships between data and the semantics behind data that are different at the enterprise level. Such system level properties will not exist in the application data models, and so a union of those models will not be sufficient for enterprise-level needs.
- **Fine-grained service calls across the network:** Aside from creating excess message overhead, invoking fine grained methods from networked services suggests that the design is focused on the technology capabilities and not the business capabilities. When business processes change, it will be difficult to update the system because so many operations will have to be modified. It is an indicator that the design used the structured, not OO, approach.

5.8 Extensibility

Extensibility can be a requirement at both the component and system level. On the component level, it is an enabler of reusability in the sense that an extensible component has design points that allow additional capability to be added to the component without major modifications. On the system level, extensibility may be affected by component extensibility but its more important aspect is the system's hospitality to the inclusion of new components to extend the larger set of capabilities of the system.

5.8.1 Enablers

- **Interoperability enablers**
- **Component Level – Clearly Defined Points of Variability:** Variability can be implemented with configuration parameters, or by programming “hooks” that allow it to call variable services, or to which new function implementations can be attached (such as OO polymorphism).
- **Plug-in Architecture:** A plug-in framework, that permits the creation of new capabilities that integrate directly into an existing component based on clearly defined plug-in interfaces and interaction protocols with the core, is a quintessentially extensible architecture.
- **System Level – Layered Architecture:** (See 5.3above)
- **System Level – Loose Coupling Between Components** (See 5.6.1above)

5.8.2 Inhibitors

- **Interoperability Inhibitors**
- **Undocumented design and architecture assumptions:** To extend a capability often involves pushing a component or system beyond where the original architectural vision. As a result, the assumptions behind many design decisions may no longer be valid. Not knowing those assumptions can make it difficult to implement the extensions or to retain correct behavior of the original functionality.

5.9 Reusability

5.9.1 Enablers

- **Extensibility enablers**
- **Use of Reusable Asset Specification (RAS):** The RAS specification is an OMG standard that is implemented in RAS creation and repository tools. Use of the specification helps asset creators understand what meta-information is needed about an asset to make it more easily reused; a searchable repository makes it more likely that analysts, architects, software designers, programmers and others can find assets that are at the right level of detail for their task(s).
- **Low code complexity:** Complex code, no matter how well modularized or tested, is more likely to have subtle problems when it is used in a different runtime context.
- **Components depend primarily on Open Architecture interfaces:** No component can work without some dependencies, so where those dependencies exist there should be to components with OA Interfaces. This increases the ease with which a reusable component can be integrated into a new environment.

5.9.2 Inhibitors

- **Extensibility inhibitors**
- **Serialization/Single-threaded implementation:** Reusable, executable assets are likely to be used in many different runtime contexts, some of which are likely to be multi-threaded. Components that are single-threaded, or have a built-in serialization are likely to be of less value. Unfortunately, it is rare that the property of a component or system is documented.
- **Violations of the Open Standards Requirement.**
- **Cut and Paste Programming:** Though a very common technique, it creates chaos in change management, and is generally considered the degenerate form of reuse [5].

5.10 Composability

5.10.1 Enablers

- **Reusability enablers**

- **Enterprise Standard Ontology:** (See 5.3 above). A rigorous, semantically consistent organizational ontology is critical to being able to compose new capabilities from existing services without change to the implementation of the services.
- **Enterprise Service Bus:** (See 5.3.1 above). Use of messaging or event driven architecture provides a profound decoupling of consumer and producer, and in conjunction with an Enterprise Standard Ontology realized in an enterprise data model, provides a solid foundation to being able to compose new capabilities from existing services.
- **Clear Qualities of Service required and provided:** In addition to a consistent model of organizational information, the qualities of service need to be clearly expressed so that appropriate services can be utilized. These can include a wide range of runtime requirements (e.g. speed of response, availability, processor, disk, connectivity, pooling, cache, interrupts, ports) and systems management (e.g. configuration, deployment, startup, shutdown, updates, problem determination, management tooling).
- **Tooling specific to composing services:** If it is possible for business users to compose services into new capabilities, it would be valuable to have tools that hide complexity, are integrated with a change management system, can track critical usage and dependency information for Enterprise Architecture management.

5.10.2 *Inhibitors*

- **Reusability inhibitors**
- **Lack of Enterprise Architecture management of Composable Services:** Composable services must be under a change control process that involves an EA approval process.
- **Intra-layer dependencies:** In a layered architecture, dependencies between components that exist at the same layer, e.g. the business component layer, increases the size of a unit of reuse and is directly in conflict with the need for decoupling. To reuse one component requires bringing along the other (and its lower layer dependencies) even if it is not needed in the new context. In most cases, it indicates a need to refactor the dependency into a higher level layer of the architecture.